数据结构与算法

第 1 讲：线性表

韩文弢

清华大学

2024—2025 学年度春季学期

本讲内容

1.1 问题引入

1.1.1 一元多项式

例. 有两个一元多项式

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

$$Q_m(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_m x^m$$

求这两个多项式的

- 和 $P_n(x) + Q_m(x)$
- 差 $P_n(x) - Q_m(x)$
- 积 $P_n(x) \cdot Q_m(x)$

思考：如何在计算机中表示一元多项式？需要考虑什么因素？

1.1.2 方法一

用数组 a 连续存储多项式 $P_n(x)$ 的系数，数组的每个分量 a[i] 表示 x^i 项的系数 a_i。 例. $P_5(x) = 1 - 3x^2 + 4x^5$

i	0	1	2	3	4	5
a[i]	1	0	−3	0	0	4

- 优点：方便实现加减法操作。
- 缺点：存储诸如 $1 + 2x^{30000}$ 这样系数稀疏的多项式会出现空间浪费的情况。

1.1.3 方法二

对于系数稀疏的情况，只需要存储系数**非零**的项。

也就是将多项式看成指数和对应项系数二元组 (k, a_k) 的集合。 例. $P_5(x) = 1 - 3x^2 + 4x^5$

i	0	1	2
a[i].exp	0	2	5
a[i].coef	1	−3	4

- 优点：对于系数稀疏的情况能节省存储空间。
- 缺点：加减法实现复杂。

1.1.4 启示

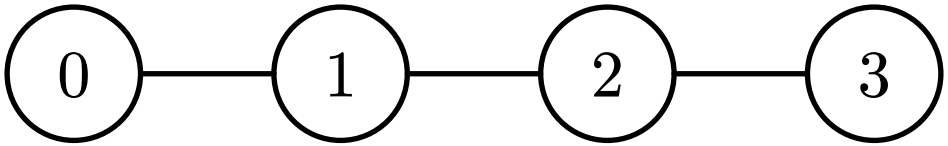
程序设计的核心任务是

- 需要在算法的简洁、可读性与时间、空间效率之间进行权衡；
- 要针对具体问题选择合适的数据结构，设计相应的算法。

1.2 线性表

1.2.1 线性表的定义

定义 2.1.1. 线性表是由同一类型的数据元素构成的线性有序的结构。没有元素的线性表称为**空表**，线性表中元素的个数称为**长度**。除空表外，线性表中有唯一的**表头**元素和唯一的**表尾**元素。除表头元素和表尾元素，线性表中每个元素都有唯一的**前驱**和唯一的**后继**。



1.2.2 线性表的抽象数据类型

伪码 1.1 : 线性表的 ADT

元素：$D = \{a_i \mid a_i \in T, i = 0, 1, 2, ..., n-1\}$，$T$ 是元素全集，n 是元素个数

关系：$R = \{(a_i, a_{i+1}) \mid a_i, a_{i+1} \in T, i = 0, 1, 2, ..., n-2\}$

基本操作：

> Initialize(L): 初始化一个空的线性表 L
>
> Destroy(L): 销毁线性表 L
>
> Clear(L): 清空线性表 L
>
> IsEmpty(L): 返回线性表 L 是否为空
>
> Length(L): 返回线性表 L 中的元素个数
>
> Get(L, p): 获得线性表 L 中位置为 p 的元素
>
> Search(L, x): 在线性表 L 中查找元素 x，返回它所在的位置，如果没有找到则返回 $\varnothing$
>
> Insert(L, p, x): 在线性表 L 的位置 p 上插入元素 x
>
> Remove(L, p): 删除线性表 L 中位置为 p 的元素

1.2.3 线性表的结构

线性表的 ADT 描述的是线性表的逻辑结构，在计算机中实现时还要设计它的存储结构，将逻辑结构映射到计算机内存中。

线性表的实现方式主要有：

- **顺序表**：元素被顺序地存储在连续的内存空间中，前驱和后继元素在物理空间上是相邻的。
- **链表**：使用指针等方式来表示元素的前驱和后继关系，不需要在物理空间上连续存储。

1.3 顺序表

伪码 1.2 : 线性表的顺序表实现

SequentialList:

> data: 指向数据的指针，数据连续存储
>
> size: 当前表中实际的元素个数
>
> capacity: 分配的容量（最多可以存储的元素个数）

1.3.2 顺序表：初始化

伪码 1.3： 初始化顺序表

```
Initialize(L):
1   L.data ← Allocate(初始容量)
2   L.size ← 0
3   L.capacity ← 初始容量
```

伪码 1.4：销毁顺序表

```
Destroy(L):
    Free(L.data)
    L.data ← ∅
    L.size ← 0
    L.capacity ← 0
```

1.3.4 顺序表：清空

伪码 1.5 : 清空顺序表

```
Clear(L):
1   | L.size ← 0
```

1.3.5 顺序表：判空

伪码 1.6 : 判断顺序表是否为空

```
IsEmpty(L):
1   return L.size = 0
```

1.3.6 顺序表：长度

伪码 1.7：获得顺序表的长度

```
Length(L):
1   return L.size
```

1.3.7 顺序表：获得元素

伪码 1.8 : 获得指定位置的元素

```
Get(L, p):
1   if ¬(0 ≤ p < L.size)
2       error 非法的访问位置
3   return L.data[p]
```

1.3.8 顺序表：查找

伪码 1.9 : 在顺序表中查找元素

```
Search(L, x):
1   for i ∈ [0, L.size)
2       if L.data[i] = x
3           return i
4   return ∅   # 未找到
```

1.3.9 顺序表：插入

伪码 1.10 : 向顺序表插入元素

Insert(L, p, x):

```
1  if ¬(0 ≤ p ≤ L.size)
2      error 非法的插入位置
3  if L.size = L.capacity
4      Expand(L)  # 增容，思考：增加多少?
5  for i ← L.size − 1 downto p
6      L.data[i + 1] = L.data[i]
7  L.data[p] ← x
8  L.size ← L.size + 1
```

1.3.10 顺序表：删除

伪码 1.11 : 从顺序表删除元素

```
Remove(L, p):
1   if ¬(0 ≤ p < L.size)
2       error 非法的删除位置
3   for i ← p to L.size − 1
4       L.data[i] = L.data[i + 1]
5   L.size ← L.size − 1
6   # 思考：什么时候调整容量?
```

1.3.11 C++ 中的顺序表 vector

C++

```cpp
#include <vector>

// 定义在 std 名字空间中，以下仅列出常用功能
template<class T>
class vector {
public:
  // 类型
  using value_type = T;
  using pointer = /* ... */;       // 以及 const_pointer
  using reference = value_type&;   // 以及 const_reference
```

1.3.11 C++ 中的顺序表 vector

```
11      using size_type = /* ... */;      // 以及 difference_type
12      using iterator = /* ... */;       // 以及 const_iterator 等
13
14      // 构造、复制、销毁
15      vector();
16      explicit vector(size_type n);  // 避免单参数隐式构造
17      vector(size_type n, const T& value);
18      template<class InputIter>
19      vector(InputIter first, InputIter last);
```

1.3.11 C++ 中的顺序表 vector

```
20    vector(const vector& x);
21    vector(initializer_list<T> il);  // 支持初始化列表
22    ~vector();
23    vector& operator=(const vector& x);
24    vector& operator=(initializer_list<T> il);
25    template<class InputIter>
26    void assign(InputIter first, InputIter last);
27    void assign(size_type n, const T& u);
28    void assign(initializer_list<T> il);
29
30    // 迭代器
```

1.3.11 C++ 中的顺序表 vector

```
31    /*const_*/iterator begin()/* const*/;
32    /*const_*/iterator end()/* const*/;
33    /*const_*/reverse_iterator rbegin()/* const*/;
34    /*const_*/reverse_iterator rend()/* const*/;
35
36    // 容量
37    bool empty() const;
38    size_type size() const;
39    size_type capacity() const;
40    void resize(size_type sz);
41    void resize(size_type sz, const T& c);
```

1.3.11 C++ 中的顺序表 vector

```
42
43      // 元素访问
44      /*const_*/reference operator[](size_type n)/*
        const*/;
45      /*const_*/reference at(size_type n)/* const*/;
46      /*const_*/reference front()/* const*/;
47      /*const_*/reference back()/* const*/;
48
49      // 数据访问
50      /*const */T* data()/* const*/;
51
```

1.3.11 C++ 中的顺序表 vector

```
52    // 修改操作
53    void push_back(const T& x);
54    void pop_back();
55
56    iterator insert(const_iterator pos, const T& x);
57    iterator insert(const_iterator pos, size_type n,
58                    const T& x);
59    template<class InputIter>
60    iterator insert(const_iterator pos, InputIter first,
61                    InputIter last);
62    iterator insert(const_iterator pos,
```

1.3.11 C++ 中的顺序表 vector

```
63                          initializer_list<T> il);
64     iterator erase(const_iterator pos);
65     iterator erase(const_iterator first, const_iterator
       last);
66     void swap(vector&);
67     void clear();
68   };
```

1.3.12 vector 使用示例

C++

```cpp
#include <iostream>
#include <vector>

int main() {
    // 创建一个整型向量
    std::vector<int> v = {8, 4, 5, 9};
    // 添加两个整数
    v.push_back(6);
    v.push_back(9);
    // 修改元素的值
    v[2] = -1;
    // 打印向量
    for (int n : v)
```

1.3.12 vector 使用示例

```
14          std::cout << n << ' ';
15      std::cout << '\n';
16  }
```

1.3.13 迭代器

C++ 中的**迭代器**是对指针的一般化抽象，用来指向**容器**中的元素，通过移动迭代器可以表示元素之间的逻辑结构。 支持的操作包括：

- `++it`：向后移动一个元素
- `--it`：向前移动一个元素
- `*it`：访问迭代器所指向的元素
- `it + n`：移动 n 个元素，需要迭代器支持随机访问
- 比较相等，比较大小（需要迭代器支持随机访问）

1.3.14 范围

C++ 中经常使用一对迭代器来表示一个区间范围 $[\text{begin}, \text{end})$。

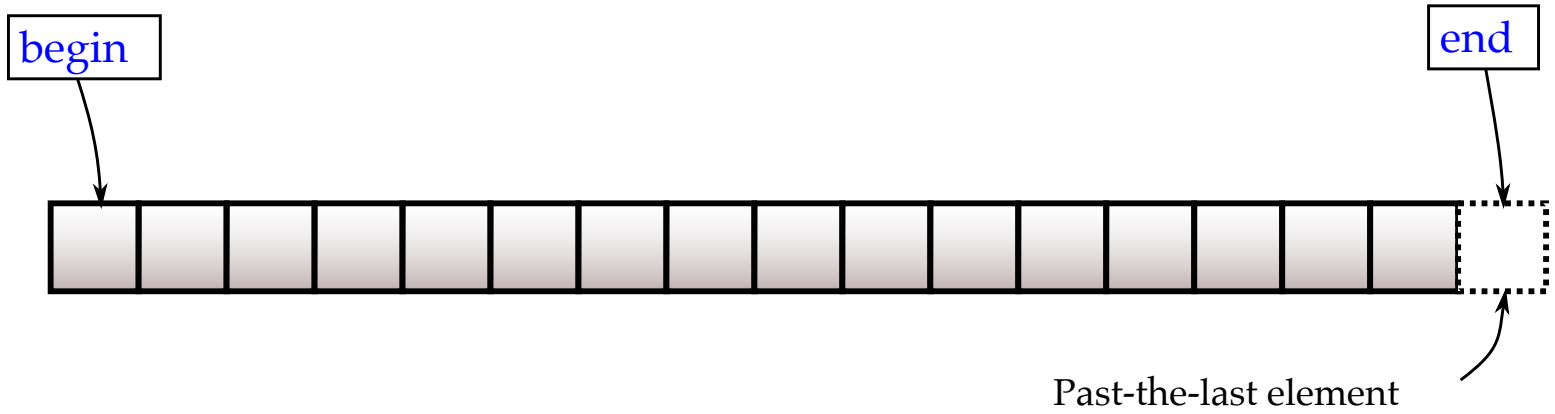


图 1.2　用迭代器表示范围

1.4 链表

1.4.1 链表的结构

顺序表的优点是可以随机访问元素，缺点是插入和删除操作需要进行元素移动。如何应对?

不使用存储位置来表示逻辑结构，给每个元素增加指针域，显式表示逻辑结构。

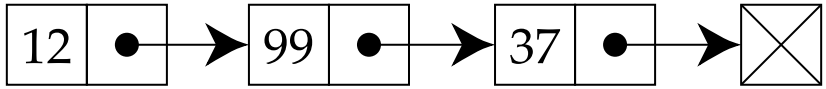


图 1.3　单向链表

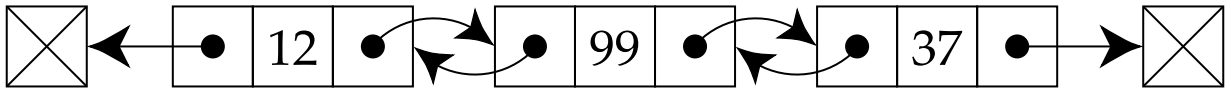


图 1.4　双向链表

1.4.2 C++ 中的链表 list

```cpp
#include <list>

template<class T>
class list {
public:
    // 类型同 vector，略
    // 构造、复制、销毁同 vector，略
    // 迭代器同 vector，略
    // 容量，没有 capacity 函数，其他同 vector
    // 元素访问，没有 operator[] 和 at 函数，其他同 vector

    // 修改操作，支持 vector 的所有操作，以及如下操作
    void push_front(const T& x);
```

1.4.2 C++ 中的链表 list

```
14    void pop_front();
15
16    // 独有的列表操作
17  };
```

1.4.3 C++ 其他线性容器

- `deque`：双端队列，采用分块存储结构，支持在两端高效增删元素。
- `array`：对普通数组的封装。
- `forward_list`：单向链表，每个元素可节省一个指针的空间，只能单向迭代。

1.5 线性表的应用

1.5.1 约瑟夫问题

问题. n 个人围成一圈，从第 1 个人开始，按顺时针方向 1 至 m 报数，报到 m 的人出列。然后从下一个人开始继续上述过程，直到剩下一个人为止。此人获得胜利，求他在最开始的圈中的位置。

例如，给出 $n = 8$，$m = 3$，则第 7 个人获胜。 解题思路：用线性表表示状态，模拟报数过程。

思考：应该选择哪种线性表?

- 顺序表：报数可用 $O(1)$ 直接定位，删除最坏情况是 $O(n)$。
- 链表：报数顺序进行是 $O(m)$，删除可在 $O(1)$ 内完成。

1.5.2 约瑟夫问题：使用 vector

C++

```cpp
#include <iostream>
#include <vector>

int main() {
    int n, m;
    std::cin >> n >> m;
    // 初始化人员序列
    std::vector<int> a(n);
    int c = 0;
    for (int& x : a) {
        x = ++c;
```

1.5.2 约瑟夫问题：使用 vector

```
12    }
13    // 报数过程
14    auto it = a.begin();
15    for (int i = 1; i < n; i++) {
16      for (int j = 1; j < m; j++) {
17        ++it;
18        // 处理到达序列末尾的情况
19        if (it == a.end()) {
20          it = a.begin();
21        }
22      }
```

1.5.2 约瑟夫问题：使用 vector

```
23        std::cout << *it << ' ';
24        // 删除报 m 的人
25        it = a.erase(it);
26        // 处理删除后变成末尾的情况
27        if (it == a.end()) {
28          it = a.begin();
29        }
30      }
31      std::cout << a.front() << std::endl;
32  }
```

1.5.3 约瑟夫问题：使用 list

C++

```cpp
#include <iostream>
#include <list>

int main() {
    int n, m;
    std::cin >> n >> m;
    // 初始化人员序列
    std::list<int> a(n);
    int c = 0;
    for (int& x : a) {
        x = ++c;
```

1.5.3 约瑟夫问题：使用 list

```
12    }
13    // 报数过程
14    auto it = a.begin();
15    for (int i = 1; i < n; i++) {
16      for (int j = 1; j < m; j++) {
17        ++it;
18        // 处理到达序列末尾的情况
19        if (it == a.end()) {
20          it = a.begin();
21        }
22      }
```

1.5.3 约瑟夫问题：使用 list

```
23        std::cout << *it << ' ';
24        // 删除报 m 的人
25        it = a.erase(it);
26        // 处理删除后变成末尾的情况
27        if (it == a.end()) {
28          it = a.begin();
29        }
30      }
31      std::cout << a.front() << std::endl;
32  }
```

1.5.4 性能比较

对于 $n = 10^6$，$m = 499$ 的情况，上述两段代码的运行情况[1]如下：

实现版本	运行时间（s）	占用内存（MB）
`vector`	19.982	6
`list`	2.669	32

[1]运行环境： Apple M1 Max 芯片的 MacBook Pro，编译器 clang 16.0.0

1.6 小结

1.6.1 本讲小结

- 线性表的定义与基本操作
- 顺序表实现
- 链表实现
- C++ 中的线性表：`vector` 和 `list`
- 线性表的应用：多项式运算、约瑟夫问题